New FORTRAN 2003 features for ABINIT 8

During the 6th ABINIT International Developer meeting (April 2013 in Dinard, France), it has been decided to allow the use of a subset of FORTRAN 2003 features in ABINIT. The present document is a first proposal of such a set of F2003 features; it is submitted for discussion.

To choose this set of features, we tried to respect the following rules:

- The F2003 features must be supported by a large majority of recent compilers. An exhaustive list of compilers and their F2003 features can be found at: http://fortranwiki.org/fortran/show/Fortran+2003+status
- We have focused on the features that are of significant importance for high-performance computing and on the language extensions that help improve the readability and the portability of the code

This is a first attempt; other F2003 extensions are not admitted for the moment, but their inclusion could be taken into consideration in the next versions. Important: introduction of the new F2003 extensions implies that, from now, not all compilers can be supported by ABINIT (see first section of the document).

M. Torrent (CEA-Bruyères-le-Châtel)
M. Giantomassi (UC Louvain)

I. Notes on Fortran compilers and ABINIT

- All the new features proposed hereafter have already been tested on the ABINIT test farm. If someone wants ABINIT to be compatible with another computing environment, a new *buildbot* slave should be added to the test farm
- Some of these extensions are **mandatory** in the sense that it won't be possible to compile the code if the compiler does not support these F2003 features (e.g. ALLOCATABLE arrays in data types). Mandatory extensions are presented in Section II.
- Other extensions are **optional** since they can be made available in a portable way via CPP macros and/or wrapper functions so that code semantics is preserved (e.g. CONTIGUOUS attribute). Optional extensions are presented in Section III.

We propose to divide the compilers in the following four categories:

1. **Officially supported**
   All tests are OK, the most important libraries and plugins are enabled and tested.

2. **Deprecated**
   Still officially supported but users are strongly encouraged to upgrade to a more recent version

or to use one of the officially supported compilers since future releases of ABINIT may drop support for this version of the compiler.
Example: *g95* (no more maintained)

3. **Unstable**
   We cannot ensure that all the features of ABINIT and the most important plugins and libraries work as expected with this version of the compiler.
   Example: *PGI v9+*

4. **Unsupported**
   Due to the introduction of the F2003 extensions, some old compilers will be excluded *de facto*.
   Example: *gfortran* versions before 4.3, *ifort* versions before 8.x, *open64*, *ORACLE*

II. Fortran 2003 features allowed in ABINIT

Mandatory extensions, implemented in (almost) all recent compilers

ALLOCATABLE arrays inside user-defined datatypes and as routine arguments

This was not possible in F95 but it is perfectly legit in F2003.
Note that ALLOCATABLE arrays are much more efficient than pointers because:

1. An allocatable array is always continuous in memory and the compiler can perform important optimizations that are not possible when pointers are involved. Pointers, indeed, may point to non-contiguous memory locations, and the compiler MUST take this possibility into account thus leading to suboptimal code.
2. The initial status of a pointer is `undefined` and this forces the developers to add a lot of boilerplate code just to nullify the pointers declared in a data type. The use of allocatable arrays solves this.
3. Runtime errors are much easier to detect and debug.

Example of a user-defined data type with an allocatable entity:

```
type (pawtab_type)
  real(dp),allocatable :: phi(:, :)
end type pawtab_type

type(pawtab_type) :: pawtab

!Allocate memory
allocate(pawtab%phi(300, 5))

!Deallocate memory
```

```
if (allocated(pawtab%phi)) deallocate(pawtab%phi)
```

As a rule of thumb, avoid pointers as much a possible. Use allocatable arrays if you need dynamic memory in a datatype or if you want to allocate memory inside a procedure and return it to the caller.

Example of a procedure that allocates memory and returns it to the caller:

```
subroutine foo(out_array)

 integer,allocatable,intent(out) :: out_array(:)

 !Compute the size of out_array
 n = ...

 !Allocate contiguous memory
 allocate(out_array(n)
 out_array = ...

end subroutine foo
```

INTENT attribute for pointers

Following the ABINIT coding rules, each argument of routine should have an `INTENT` attribute, even pointers. This is now possible in F2003.

Example:

```
real(dp), pointer, intent(in) :: cg_ptr(:,:)
```

Note that the intent refers to the association status of the pointer without any reference to the target. If you are not sure about which intent should be used, use `intent(in)`.

Access to computing environment

F2003 gives access to a lot of information about the computing environment.

Example:

```
CALL GET_ENVIRONMENT_VARIABLE(…) ! to get an environment variable
CALL GET_COMMAND(..)
CALL COMMAND_ARGUMENT_COUNT(..)
CALL GET_COMMAND_ARGUMENT(,…) ! to access to the command line
```

These features can be used to improve the user interface (e.g. command line options can be passed to ABINIT), or to modify the behavior at run-time depending on the value of the environment variables.

Array constructor with [ ] syntax

This simple extension makes the array constructor syntax more readable. Developers are encouraged to use the new syntax whenever possible in order to improve the readability of the code.

Example :

```
integer, parameter :: cute(4)=[1,2,3,4]
```

Named constants in complex constant declaration

A very simple – but useful – feature.

Example:

```
complex(dpc), parameter :: CC=(0._dp,pi)
```

Use of interoperability with C

The F2003 intrinsic module `ISO_C_BINDINGS` provides a standardized interface that facilitates the interoperability between Fortran and C. This new feature is of paramount importance for high-performance computing since one can:

- Call C procedures from Fortran (e.g. FFTW3),
- Pass complex arguments to Fortran functions having an explicit interface with reals without having to perform a detrimental copy-in, copy out.

For this, the availability of the `ISO_C_BINDINGS` module is required; this is OK for all recent compilers.

At present, ABINIT requires a `ISO_C_BINDINGS` module providing:

- C types: `C_INT`, `C_SHORT`, `C_LONG`, etc…
- C pointer `C_PTR` type
- `C_LOC` and `C_F_POINTER` functions.
  `C_LOC` returns the location of a C pointer associated to a Fortran object.
  `C_F_POINTER` returns a Fortran pointer from a C pointer.
- `C_BIND` attribute that allows the developer to rename a C function as Fortran one or to define a Fortran datatype that is interoperable with a C structure.

These features are already used in important parts such as CUDA or FFTW3.

Example (how to call a C function):

```
type(C_PTR) :: blabla
```

```
complex(C_INT) :: data
blabla = my_c_function(data)
```

Example:
How to avoid a copy while passing real arrays to a procedure that expects complex arguments and has an explicit interface
Copy *real* array `data1_real` in `data2_real` using *complex* function `zcopy`:

```
real(dp),target :: data1_real(2,nn),data2_real(2,nn)
complex(dpc), ABI_CONTIGUOUS pointer :: data1_cplx(:),data2_cplx(:)

call C_F_POINTER(C_LOC(data1_real),data1_cplex, shape=[nn])
call C_F_POINTER(C_LOC(data2_real),data2_cplex, shape=[nn])

!If zcopy has an explicit interface, the compiler won't
!allow you to pass data1_real because the procedure expects
!an argument of type complex(dpc)

call zcopy(nn,data1_cplx,data2_cplx)
```

We propose to hide all this stuff in a «helper function» that will return a complex Fortran pointer from a real Fortran array.

IMPORT statement

Used in interfaces, it allows the access to the variables and the types defined in modules.

Example:

```
use m_pawtab, only : pawtab_type

interface
  subroutine my_rout(pawtab)
  IMPORT :: pawtab_type
  type(pawtab_type), intent(inout) :: pawtab
end interface
```

III. Fortran 2003 features allowed in ABINIT only by the use of specific functions

Optional extensions, implemented in the majority of compilers, but not all

Flush statement

The execution of a FLUSH statement for an external file causes data written to it to be available to other processes.

One should flush Fortran files though the `flush_unit` helper function or by calling `wrtout` with the optional argument `do_flush`.
The reason for such limitation is that not all the compilers provide a standard Fortran `flush` and `flush_unit` uses CPP options to implement this feature in a semi-portable way.

Example:

```
open(unit=unt, file="foo.dat")
write(unt, *)"hello world"
call flush_unit(unt)

!For flushing ab_out or std_out, use
call wrtout(std_out,message,'COLL',do_flush=.true.)
```

Note that the `flush` statement is very important to avoid race conditions that can occur when multiple `MPI` processors are reading and writing from the same file.

Example:

```
open(unit=unt, file="foo.dat")

if (me==master) then
  write(unt,*)"hello"
  !Flush the unit so that we force the writing of the string
  !that will be read by the other nodes
  call flush_unit(unt)
end if

call xmpi_barrier(comm)
if (me /= master) then
  !Can safely read the string because master called flush_unit
  read(unt,*)string
end if
```

The `MPI` library provides an interface to flush the IO buffer but only if the file has been open with `MPI_FILE_OPEN`.

IOMSG and NEWUNIT specifiers in a OPEN statement

`IOMSG` is a string with a description of the error that may occur during an IO operation (e.g not enough permissions, disk quota error, non-existent file when reading ...)

`NEWUNIT` is used to receive a free unit, i.e. a logical Fortran unit that is not already in use in other

parts of the code.

Developers should take advantage of these new features through the `open_file` helper function defined in the `m_io_tools` module.

Example:

```
use m_io_tools, only : open_file

ii = open_file(filename,iomsg,newunit=tmp_unit, &
&              form="formatted",action="write")

!If IOError, print a message describing the error and stop the code
if (ii/=0) MSG_ERROR(iomsg)
```

Use of IEEE Arithmetic

This extension allows one to trap or to signal possible floating point exceptions at run time without compiling the code in debug mode. Developers can use the procedures defined in the `m_xiee.F90` module for debugging/testing purposes (these functions are empty if the compiler does not support IEEE arithmetic).

Production code should not contain debugging sections with calls to the procedures defined in `m_xieee.F90` in order to avoid a significant slowdown of the code. Note that one can easily trap/signal floating-point exceptions thanks to the *command line options* `–ieee-halt` and `–ieee-signal`.

Fortran extensions supported via CPP macros

This section discusses a set of features belonging to F2003/F2008 that are vey useful for optimizing CPU-critical parts or for constructing reliable software. The features described in this section are made available in a portable way via CPP macros defined in `abi_common.h`.

The most important attributes supported at the time of writing are:

- **`CONTIGUOUS`** attribute, emulated by **`ABI_CONTIGUOUS`** macro,
- **`ASYNCHRONOUS`** attribute emulated by **`ABI_ASYNCH`** macro,
- **`PROTECTED`** attribute, emulated by **`ABI_PROTECTED`** macro,
- **`PRIVATE`** attribute, emulated by **`ABI_PRIVATE`** macro.

The `CONTIGUOUS` attribute is principally used for pointers that are used to point contiguous portions of memory. `PROTECTED` and `PRIVATE` are mainly used for data hiding and the design of libraries and robust code.

Example:

```
integer, ABI_CONTIGUOUS pointer :: ptr(:)
```